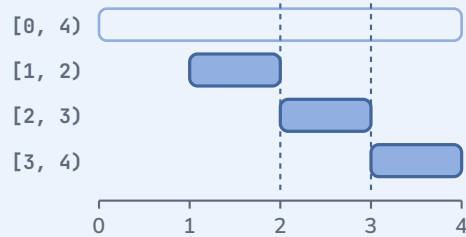


Greedy methods

Chapter 2.5 · Overall level: Core



A greedy method builds an answer one decision at a time, always taking the choice that looks best right now and never going back. *When can such a short-sighted rule be trusted to give the best answer overall?*

Sometimes it can, and then the program is short and fast: sort, then make one pass. Often it cannot, and a plausible rule quietly gives wrong answers. This chapter is about telling the two apart: finding a counterexample when a rule is wrong, and writing an exchange argument when it is right.

2.5.1 Choosing sessions one at a time

A school has one computer room and a list of requests for it. Each request is a session $[s, e)$ that starts at time s and ends at time e . The round bracket at the end is deliberate: the session uses the room from s up to, but not including, e . Such an interval is called **half-open**, and it means a session ending at 3 and another starting at 3 do not clash. The organiser wants to accept as many sessions as possible, with no two overlapping.

Trying every subset of n sessions means 2^n subsets, hopeless for $n = 200,000$. A greedy method instead fixes a rule for which session to accept next, accepts it, discards everything that clashes with it, and repeats. Three rules come to mind quickly:

- **Shortest first.** A short session uses little of the room, so it should leave the most room for others.
- **Earliest start first.** Fill the room from the beginning of the day, leaving no early gaps.
- **Earliest finish first.** Free the room again as soon as possible.

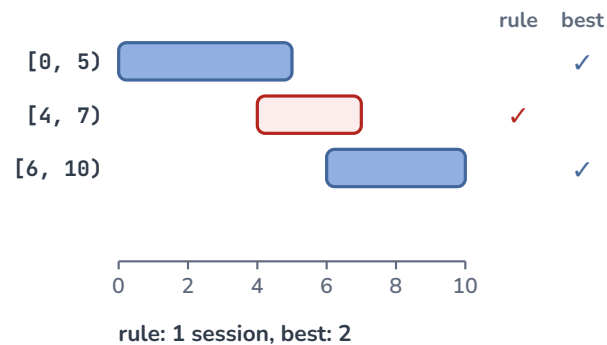
All three sound sensible, and all three give the right answer on many inputs. At most one of them can be right on every input, and the quickest way to rule a rule out is to find one small input where it loses.

2.5.2 Counterexamples to plausible rules

A **counterexample** is a single input on which a rule gives a worse answer than the best possible one. One counterexample is enough to reject a rule for good, however well it does elsewhere. Good counterexamples are tiny, so that anyone can check them by hand.

For shortest first, take $[0, 5)$, $[4, 7)$ and $[6, 10)$. The shortest session is $[4, 7)$, of length 3, but it overlaps both of the others, so the rule ends with one session, while $[0, 5)$ and $[6, 10)$ fit together. For earliest start first, take $[0, 4)$, $[1, 2)$, $[2, 3)$ and $[3, 4)$. The rule grabs $[0, 4)$ first, and that one long session blocks three short ones that would all fit.

Shortest first



Earliest start first

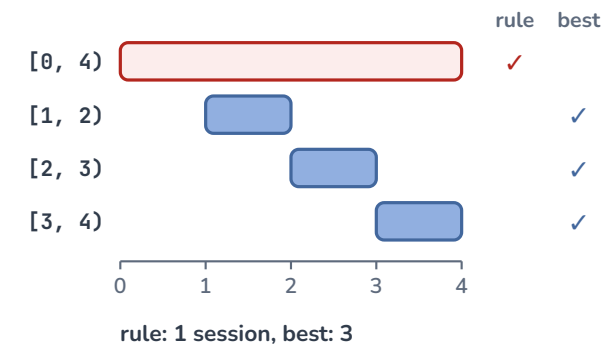


Figure 2.5.1 — Two small inputs that defeat two plausible rules. Red bars and the “rule” column show what the rule picks; blue bars and the “best” column show an optimal choice.

2.5.3 Earliest finish first

The third rule survives. Sort the sessions by end time, go through them in that order, and accept a session whenever it starts no earlier than the end of the last accepted one. On the four sessions above it accepts $[1, 2)$, then $[2, 3)$ because it starts exactly when $[1, 2)$ ends, skips $[0, 4)$, and accepts $[3, 4)$: three sessions, the best possible.

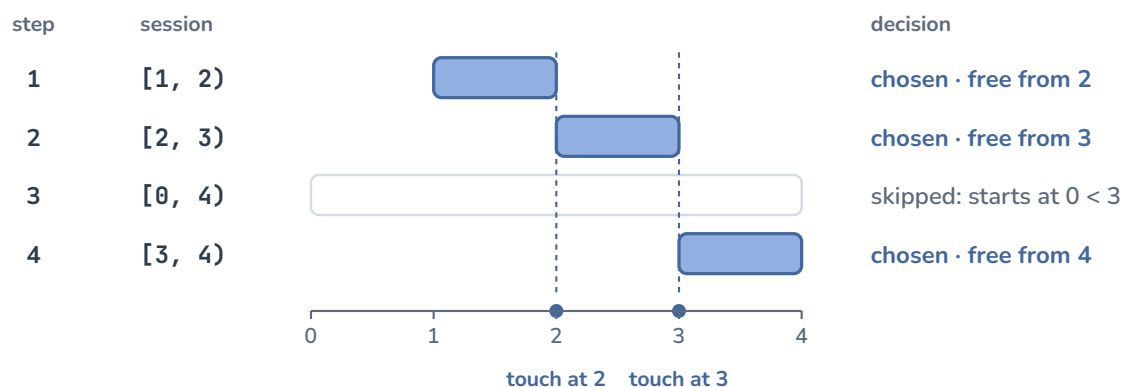


Figure 2.5.2 — Finishing earlier leaves at least as much room for every later choice. Rows are in the order the rule examines them; the dashed lines mark touching endpoints, which half-open sessions allow.

The program needs only one number besides the sorted list: `free_from`, the time from which the room is free. It starts at 0, because no session starts before time 0. A session is accepted when $s \geq \text{free_from}$, and then `free_from` becomes its end. Here is the trace for the sample of the first practice problem: sessions $[0, 3)$, $[1, 2)$, $[2, 4)$, $[3, 5)$; it accepts two of them, as expected.

Step	Session (by end)	free_from before	Decision	free_from after
1	$[1, 2)$	0	accept: $1 \geq 0$	2
2	$[0, 3)$	2	skip: $0 < 2$	2
3	$[2, 4)$	2	accept: $2 \geq 2$ (touching)	4
4	$[3, 5)$	4	skip: $3 < 4$	4

2.5.4 Why earliest finish is optimal: an exchange argument

A rule that passes three test cases has not been proved; the counterexamples above passed plenty of tests too. A greedy rule needs an argument that covers every input. The usual tool is an **exchange argument**: start from any optimal answer and show that it can be changed, one step at a time, into the greedy answer without ever getting worse.

Let g be the session with the earliest end. Take any optimal schedule O and list its sessions in time order; call the first one o . Because g finishes earliest of all sessions, $e(g) \leq e(o)$. Now replace o by g . Every other session of O starts at or after $e(o)$, so also at or after $e(g)$: nothing clashes, and the schedule keeps its size. Therefore some optimal schedule begins with g .

Once g is accepted, what remains is the same problem on the sessions that start at or after $e(g)$, and the rule treats those exactly as it would treat a fresh input. Applying the same exchange to that smaller problem, again and again (induction on the number of sessions), shows that the greedy schedule is as large as an optimal one.

Invariant. After the rule has accepted k sessions, its k -th session ends no later than the k -th session of any valid schedule. Finishing earlier never removes an option for later, which is exactly what the caption of Figure 2.5.2 says.

Pseudocode

Earliest finish first: $O(n \log n)$ to sort, then $O(n)$

```

read n and the sessions [s, e)
sort the sessions by e
count ← 0; free_from ← 0
for each session [s, e) in that order:
    if s ≥ free_from:                (touching is allowed)
        count ← count + 1
        free_from ← e
output count

```

2.5.5 Code in four languages

Each program solves the first practice problem, **Compatible sessions**. The first line holds n ; each of the next n lines holds one session “s e”, meaning $[s, e)$. The program prints one integer, the largest number of sessions that can be chosen.

Sample input	Sample output
4 0 3 1 2 2 4 3 5	2

Sorting dominates the running time: $O(n \log n)$ time and $O(n)$ memory for the list. Times are at most 10^9 , which fits in a 32-bit `int`. Python and C++ store each pair with the end first, so the built-in sort orders by end; Java and JavaScript sort with an explicit comparison on the end time.

Python 3

```
import sys

def main():
    data = sys.stdin.buffer.read().split()
    n = int(data[0])
    sessions = []
    for k in range(n):
        s, e = int(data[1 + 2 * k]), int(data[2 + 2 * k])
        sessions.append((e, s))      # end first, so sorting orders by end
    sessions.sort()
    count = 0
    free_from = 0                    # the room is free from this time on
    for e, s in sessions:
        if s >= free_from:           # touching is allowed: [a, t) then [t, b)
            count += 1
            free_from = e
    print(count)

main()
```

C++17

```

#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    cin >> n;
    vector<pair<int, int>> sessions(n);    // (end, start)
    for (auto& [e, s] : sessions) cin >> s >> e;
    sort(sessions.begin(), sessions.end()); // by end time
    int count = 0;
    int freeFrom = 0;                    // the room is free from here on
    for (auto& [e, s] : sessions) {
        if (s >= freeFrom) {             // touching endpoints are fine
            count++;
            freeFrom = e;
        }
    }
    cout << count << '\n';
    return 0;
}

```

Java 17

```

import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) throws IOException {
        StreamTokenizer in = new StreamTokenizer(
            new BufferedReader(new InputStreamReader(System.in)));
        in.nextToken();
        int n = (int) in.nval;
        int[][] sessions = new int[n][2];    // {start, end}
        for (int k = 0; k < n; k++) {
            in.nextToken(); sessions[k][0] = (int) in.nval;
            in.nextToken(); sessions[k][1] = (int) in.nval;
        }
        Arrays.sort(sessions, (x, y) -> Integer.compare(x[1], y[1]));
        int count = 0;
        int freeFrom = 0;                    // the room is free from here on
        for (int[] ses : sessions) {
            if (ses[0] >= freeFrom) {         // touching endpoints are fine
                count++;
                freeFrom = ses[1];
            }
        }
        System.out.println(count);
    }
}

```

JavaScript (Node)

```
const data = require("fs").readFileSync(0, "utf8")
  .trim().split(/\s+/).map(Number);
const n = data[0];
const start = new Array(n), end = new Array(n), order = new Array(n);
for (let k = 0; k < n; k++) {
  start[k] = data[1 + 2 * k];
  end[k] = data[2 + 2 * k];
  order[k] = k;
}
order.sort((x, y) => end[x] - end[y]); // indices by end time
let count = 0;
let freeFrom = 0; // the room is free from here on
for (const k of order) {
  if (start[k] >= freeFrom) { // touching endpoints are fine
    count++;
    freeFrom = end[k];
  }
}
console.log(count);
```

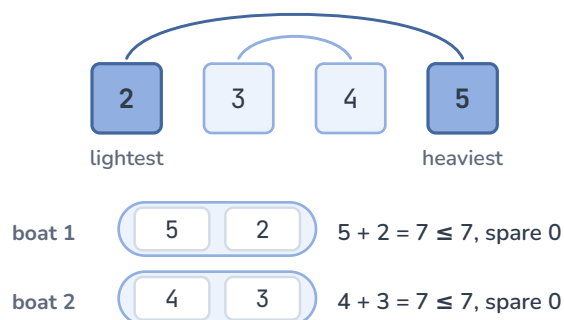
Sessions with equal end times may be examined in any order: once one of them is accepted, `free_from` equals that end, and every other session ending at the same time starts before it and is skipped.

2.5.6 Pairing passengers in two-seat boats

A different greedy problem: each boat carries at most two people, whose total weight must be at most a capacity w . Every person fits alone. What is the smallest number of boats? With weights 2, 3, 4, 5 and capacity 7, two boats suffice: $2 + 5$ and $3 + 4$.

Sort the weights. Look at the heaviest remaining person h and the lightest remaining person l . If $h + l \leq w$, they share a boat; otherwise h goes alone. Remove whoever left and repeat. Two indices, one at each end of the sorted list, move towards each other, so after sorting this is a single $O(n)$ pass (the two-pointer pattern of Chapter 2.2).

Heaviest with lightest, capacity 7



Exchange when $h + l \leq w$

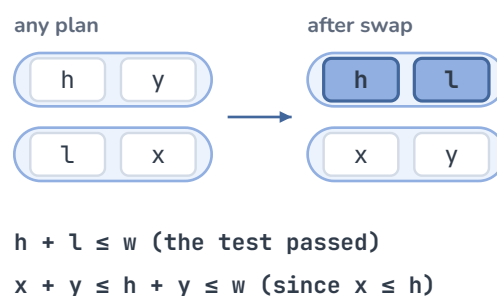


Figure 2.5.3 — If the heaviest cannot pair with the lightest, it cannot pair with anyone. If it can, any plan can be rearranged so that it does, without using more boats.

Why the heaviest person decides

If $h + l > w$, then h cannot share with anyone, because everybody else weighs at least l . So h travels alone in every possible plan, and sending h alone now loses nothing.

If $h + l \leq w$, take any optimal plan and rearrange it so that h and l share a boat, without adding boats. If they already share, there is nothing to do. If h rides with y and l rides with x , swap y and l : the boat $h + l$ fits by the test, and the boat $x + y$ fits because $x \leq h$ gives $x + y \leq h + y \leq w$. If one of them rides alone, move the other into that boat; the partner left behind rides alone. If both ride alone, putting them together even saves a boat. In every case the count does not grow, so pairing h with l is safe, and the same argument applies to the people who remain.

Step	Lightest l	Heaviest h	$h + l$	Decision	Boats so far
1	2	7	$9 > 8$	7 goes alone	1
2	2	6	$8 \leq 8$	6 and 2 share	2
3	3	4	$7 \leq 8$	4 and 3 share	3

The table traces weights 2, 3, 4, 6, 7 with capacity 8. Three boats are also clearly necessary: five people need at least three two-seat boats.

Compare a tempting alternative: “put the two lightest people together, since they surely fit”. With weights 1, 2, 5, 6 and capacity 7 it makes the boat {1, 2}, and then 5 and 6 cannot share ($11 > 7$), so it uses three boats. Heaviest-with-lightest makes {6, 1} and {5, 2}: two boats. Light people are valuable precisely because they can partner the heavy ones, so spending them on each other wastes them.

Hunting for counterexamples with a brute force

Before trying to prove a greedy rule, it pays to attack it. Write a tiny exhaustive program that tries every possibility (every subset of sessions, every way to pair the passengers) and compare it with the greedy program on thousands of random inputs with three to eight items and small numbers. A wrong rule is usually caught within seconds, and the failing input is small enough to study by hand; that is how inputs like those in Figure 2.5.1 are found. If no difference appears, that is encouraging but still not a proof: the exchange argument is.

2.5.7 Merging intervals and the length of a union

Now the sessions are not competing for a room; they record when a machine was in use, and the question is how much time was *covered* at least once. Overlapping parts must count only once. Adding up the lengths overcounts, and marking every time unit is too slow when times reach 10^9 .

Sort the intervals by **start** and sweep from left to right, keeping one current block $[cs, ce)$. For the next interval $[s, e)$:

- if $s \leq ce$, it overlaps or touches the block, so the block grows: $ce \leftarrow \max(ce, e)$;
- if $s > ce$, there is a gap. The block is finished: add $ce - cs$ to the answer and open a new block $[s, e)$.

When the list ends, add the length of the last block. The max matters: an interval lying inside the block, such as $[2, 3)$ inside $[1, 10)$, must not shrink it.

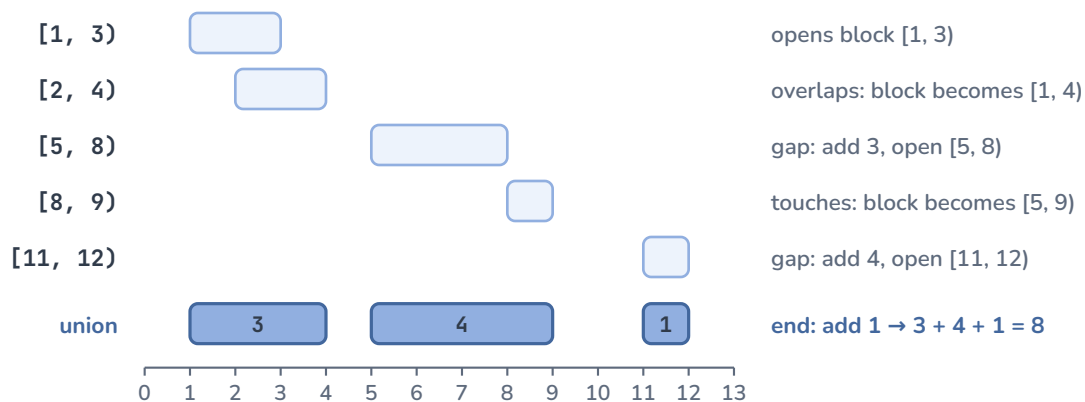


Figure 2.5.4 — A sweep in start order. Each interval either extends the current block or, after a gap, closes it for good.

Why a closed block is final. The list is sorted by start, so when a gap appears ($s > ce$), every interval still to come starts at s or later, which is beyond ce . None of them can reach back into the finished block. Inside a block there are no holes, because each interval added to it started at or before its current end. So the blocks are separate, together they cover exactly the union, and the answer is the sum of their lengths.

For the sample of the third practice problem, $[1, 4)$, $[3, 7)$, $[9, 10)$, $[7, 9)$ sorted by start become $[1, 4)$, $[3, 7)$, $[7, 9)$, $[9, 10)$. Each one overlaps or touches the block, which grows to $[1, 10)$: length 9. Sorting by end instead would be a mistake here; this sweep needs every interval that could join the current block to arrive before the block is closed. The cost is $O(n \log n)$ for sorting plus one pass.

2.5.8 A meeting point: the median

Friends live at positions $a_1, \dots, a_n$ along a straight road and want to meet at an integer point x that makes the total distance $|a_1 - x| + \dots + |a_n - x|$ as small as possible. Here the greedy idea is a local move: start anywhere and step towards the side where the step helps.

Watch what one step to the right does. Every point at or to the left of x becomes 1 farther away; every point to the right becomes 1 closer. If L points lie at or left of x and R points lie to its right, the total changes by $L - R$. So stepping right helps exactly while more points are to the right than at or to the left, and it stops helping when x reaches the middle of the sorted list: a **median**. The same reasoning applies to steps to the left.

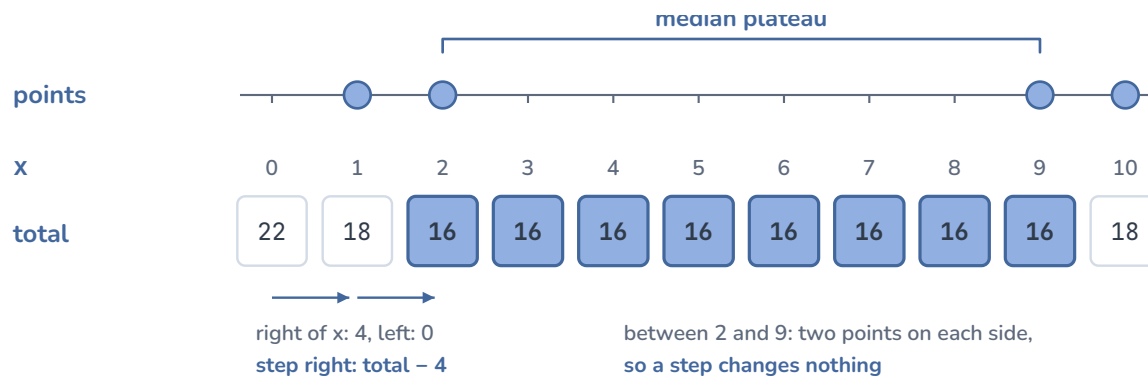


Figure 2.5.5 — Totals for every integer x on the points 1, 2, 9, 10. Each step towards the middle removes the imbalance between the two sides; every x from 2 to 9 gives the minimum, 16.

With an odd number of points the middle value is the only best point. With an even number, every x between the two middle values (inclusive) is equally good, which is why the sample of the fourth practice problem accepts any x from 2 to 9. In both cases, after sorting, the element at 0-based index $n // 2$ is a median, and the answer is the sum of distances to it: $O(n \log n)$ time.

A median balances *counts*, not sizes. For points 0, 0, 0, 100 the average is 25, with total distance 150, while the median 0 gives 100. A single far-away point pulls the average but not the median.

Size check: 200,000 points, each up to 2×10^9 away, can total 4×10^{14} . Use `long` in C++ and `long` in Java; a JavaScript Number is exact up to about 9×10^{15} , so it is safe here.

2.5.9 Recognising a greedy problem

Problem	Sort by	Greedy step	Why it is safe
Most compatible sessions	end	accept if $s \geq \text{free_from}$	exchange the first session
Two-seat boats	weight	heaviest with lightest, if they fit	swap the partners
Union length	start	extend or close the block	a closed block is final
Meeting point	value	step towards the median	compare L and R

2.5.10 Common mistakes

- **Treating closed and half-open intervals alike.** Half-open $[1, 3)$ and $[3, 5)$ are compatible; closed $[1, 3]$ and $[3, 5]$ both contain time 3 and clash. In the code this is the difference between $s \geq \text{free_from}$ and $s > \text{free_from}$. Read which kind the statement uses.
- **Greedy change-making for any coin system.** “Always take the largest coin that fits” is right for coins such as 1, 2, 5, 10, but not in general. With coins 1, 3, 4 and amount 6 it pays $4 + 1 + 1$ (three coins), while $3 + 3$ uses two. The minimum number of coins is a dynamic programming problem (Chapter 3.4).
- **Giving a heuristic as a proof.** “It is intuitive” or “it worked on my tests” is not a reason. Either find a counterexample or give an exchange argument or invariant.
- **Sorting by the wrong key.** Session selection sorts by end; merging sorts by start. Swapping them gives wrong answers on small inputs.
- **Shrinking a merged block.** Write $ce \leftarrow \max(ce, e)$, not $ce \leftarrow e$, or a contained interval cuts the block short.

2.5.11 Where greedy stops working

Greedy rules are fragile: a small change to a problem can break the exchange argument. If each session has a value and the goal is the largest total value, earliest finish fails: with $[0, 10)$ worth 10 and $[0, 1)$, $[1, 2)$ worth 1 each, it collects 2 instead of 10. That weighted version, minimum coins for an arbitrary coin system and the knapsack problem need dynamic programming (Chapters 3.4 and 3.5). Boats with three seats break the pairing argument too. Before trusting a greedy rule, try to break it on tiny inputs, then prove it.

Exercises

Difficulty: ● quick check, ●● needs an extra idea. Try each one before reading the answers.

1. ● Can the sessions $[1, 3)$ and $[3, 5)$ both be scheduled? Would the answer change for closed intervals $[1, 3]$ and $[3, 5]$?
2. ● A greedy rule gives the correct answer on three examples. Is that a proof that the rule is correct?
3. ● Run earliest finish first on $[2, 6)$, $[1, 3)$, $[5, 7)$, $[3, 5)$, $[6, 8)$. Which sessions are accepted?
4. ●● Boats have capacity 10. The weights are 9, 1, 6, 8, 5, 2. How many boats does the heaviest-with-lightest rule use, and in which pairs?
5. ●● Find the length of the union of $[4, 6)$, $[0, 2)$, $[1, 3)$, $[6, 7)$, $[9, 12)$ with the merging sweep.
6. ●● For points 3, -4 , 8, 1, 6, find the best meeting point and the minimum total distance. Why is moving from $x = 4$ to $x = 3$ an improvement?

Answers and explanations

Read these after attempting the exercises.

1. Yes. The first ends at 3 and the second starts at 3, and a half-open session does not use its end time. Closed intervals $[1, 3]$ and $[3, 5]$ both contain 3, so they clash.
2. No. Plausible wrong rules also pass many examples (Figure 2.5.1). A proof needs an argument for every input, such as an invariant or an exchange argument.
3. By end: $[1, 3)$, $[3, 5)$, $[2, 6)$, $[5, 7)$, $[6, 8)$. Accept $[1, 3)$ (free from 3), accept $[3, 5)$ (free from 5), skip $[2, 6)$ since $2 < 5$, accept $[5, 7)$ (free from 7), skip $[6, 8)$ since $6 < 7$. **Three sessions:** $[1, 3)$, $[3, 5)$, $[5, 7)$.
4. Sorted: 1, 2, 5, 6, 8, 9. $9 + 1 = 10$ fits; $8 + 2 = 10$ fits; $6 + 5 = 11$ does not, so 6 goes alone, then 5 goes alone. **Four boats:** $\{9, 1\}$, $\{8, 2\}$, $\{6\}$, $\{5\}$.
5. Sorted by start: $[0, 2)$, $[1, 3)$, $[4, 6)$, $[6, 7)$, $[9, 12)$. Blocks $[0, 3)$, $[4, 7)$ (the touching $[6, 7)$ joins) and $[9, 12)$; $3 + 3 + 3 = 9$.
6. Sorted: $-4, 1, 3, 6, 8$. The median is 3 and the total is $7 + 2 + 0 + 3 + 5 = 17$. At $x = 4$ three points ($-4, 1, 3$) lie to the left and two to the right; a step left brings three points closer and moves two farther away, so the total drops by 1, from 18 to 17.

Practice this topic

Four coding problems for this chapter are in the **Practice** tab. Open any of them in any order; the label shows difficulty within this topic.

Compatible sessions

Apply

Choose as many sessions as possible so that no chosen time intervals overlap. A session occupies $[s, e)$; one ending at time t is compatible with one starting at t .

Two-seat boats

Connect

Each boat carries at most two people with total weight at most w . Find the minimum boats needed. Every person individually fits.

Covered time

Reason

Find the total length covered by the union of n half-open time intervals. Overlaps count only once.

Meeting coordinate

Stretch

Choose an integer x minimizing $\sum |a_i - x|$. Print the minimum sum; you do not need to print x .